**DRAFT: COMMENTS SOLICITED**

# GriPhyN/PPDG
# Data Grid Architecture, Toolkit, and Roadmap
# — Version 2 —

## The GriPhyN and PPDG Collaborations

**Editors: Ian Foster, Carl Kesselman**

(foster@mcs.anl.gov, carl@isi.edu)


**Contributors: Ewa Deelman, Ian Foster, Carl Kesselman, Harvey Newman, Doug Olson, Ruth Pordes, Richard Mount, Alain Roy, Mike Wilde**

## Abstract

Data Grids are emerging as important to a range of scientific and engineering disciplines that depend for their progress on community creation, curation, and often computationally intensive analysis of large quantities of data.  GriPhyN and PPDG are two major U.S. research and development projects aimed at creating and realizing the promise of Data Grids in practical settings, emphasizing in particular the requirements of physics and astronomy applications but also working closely with other domains.  This document provides a comprehensive statement of our current understanding of the *requirements* that motivate development of Data Grids and then addresses three aspects of our approach to addressing these requirements: (1) the *architecture* that has been defined by the GriPhyN and PPDG projects to meet these requirements; (2) the current state of our instantiation of this architecture as realized in the GriPhyN *Virtual Data Toolkit* and PPDG experiment end-to-end applications, and (3) a *roadmap* that defines both the path that we expect the projects to follow to extend the reach of the toolkit and application components, and areas in which no work is currently planned (and that hence represent areas where new projects are perhaps needed).  The document is intended to be a complete treatment of these issues, although in many cases it refers to other documents for technical details concerning APIs, protocols, and requirements.  This is the second version of this document, which will evolve through a series of revisions for the foreseeable future.

# Table of Contents

# 1   Introduction

The term "Grid" refers to technologies and infrastructure that enable *coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations* [12, 14]. This sharing relates primarily to direct access to computers, software, data, networks, storage and other resources, as is required by a range of collaborative computational problem-solving and resource-brokering strategies emerging in industry, science, and engineering.

Grid concepts are particularly relevant to data-intensive science due to the collaborative nature of data production and analysis, and the increasing complexity of data analysis tasks, which, increasingly, requires the harnessing of large distributed collections of shared resources. We thus see considerable interest in the creation of so-called *Data Grid*s, i.e., Grid infrastructures, tools, and applications designed to enable distributed access to, and analysis of, large amounts of data.

Within the U.S., the NSF-funded Grid Physics Network (GriPhyN, www.griphyn.org) and the DOE-funded Earth System Grid (ESG, www.earthsystemgrid.org) and Particle Physics Data Grid (PPDG, www.ppdg.net) projects are working together to develop a comprehensive set of Data Grid technologies to be deployed on a large scale by a set of partner science projects. (In Europe, the EU DataGrid, www.eu-datagrid.org, project is pursuing similar goals.) The partner science projects targeted by GriPhyN, ESG, and PPDG are, initially, experiments in high energy and nuclear physics, astronomy, and climate science, although there are also close ties with other science and engineering disciplines.

GriPhyN and PPDG partners are engaged *now* in developing next-generation Grid-based data analysis systems. It is hence important to document clearly the requirements that the GriPhyN/PPDG software is intended to address, the overall architecture of this software, and the APIs and protocols to which the software adheres. It is also important to document planned enhancements to the software and to identify major functionality that is missing—and that hence represents opportunities for contributions by other partners, and/or future collaborative development.

This document is intended to provide in one place a comprehensive review of requirements, architecture, current state, and future plans for the GriPhyN and PPDG common software suite. In many cases, this review refers to other documents for technical details, but we aim to be complete in terms of documenting current understanding of the issues that must be addressed in a Data Grid system, and the current status of our "production" software. Note that in addition to the common software documented here, there will always be other software components that are being used by individual experiments or projects. These may eventually appear into subsequent versions of this document, if they generate widespread interest.

The Data Grid architecture has a highly modular structure, defining a variety of components, each with its own protocol and/or application programmer interfaces (APIs). These various components are designed for use in an integrated fashion, as part of a comprehensive Data Grid architecture. However, the modular structure facilitates integration with discipline-specific systems that already have substantial investments in data management systems.

This second version of this document represents a significant evolution over the first version [11]. Further versions will be produced from time to time as our software, and understanding, evolves.

# 2 Requirements

## 2.1 General Statement of Requirements

We provide here some background information, taken in part from [7], on the problems we wish to solve. The experiments targeted by GriPhyN and PPDG (ATLAS, BaBar, CMS, D0, JLAB, LIGO, SDSS, STAR, TJNF, and NVO [29]) share with other data-intensive applications a need to manage and process large amounts of data [6, 21]. This data comprises raw (measured) and many levels of processed or refined data as well as comprehensive metadata describing, for example, how the data was generated, how large it is, etc. The total size of this data ranges from terabytes (in the case of SDSS) to petabytes (in the case of ATLAS and CMS [S. Bethke (Chair), 2001 #1757]).

The computational and data management problems encountered in these experiments include the following challenging aspects:

- *Computation-intensive as well as data-intensive*: Analysis tasks are compute-intensive *and* data-intensive and can involve thousands of computer, data handling, and network resources. The central problem is coordinated management of computation and data, not just data curation and movement.

- *Need for large-scale coordination without centralized control*: Stringent performance goals require coordinated management of numerous resources, yet these resources are, for both technical and strategic reasons, highly distributed and not amenable to tight centralized control.

- *Large dynamic range in user demands and resource capabilities*: We must be able to support and arbitrate among a complex task mix of experiment-wide, group-oriented, and (thousands of) individual activities—using I/O channels, local area networks, and wide area networks that span several distance scales.

- *Data and resource sharing*: Large dynamic communities would like to benefit from the advantages of intra and inter community sharing of data products and the resources needed to produce and store them.

We introduce the *Data Grid* as a unifying concept to describe the new technologies required to support such next-generation data-intensive applications—technologies that will be critical to future data-intensive computing not only in the physics experiments mentioned above, but in the many areas of science and commerce in which sophisticated software must harness large amounts of computing, communication and storage resources to extract information from measured data. We use the term Data Grid to capture the following unique characteristics:

- A Data Grid has *large extent*—national or worldwide—and *scale*, incorporating large numbers of resources and users on multiple distance scales.

- A Data Grid is more than a network: it layers sophisticated *new services* on top of local policies, mechanisms, and interfaces, so that geographically remote resources (hardware, software and data) can be shared in a coordinated fashion.

- A Data Grid provides a new degree of *transparency* in how data-handling and processing capabilities are integrated to deliver data products to end-user applications, so that requests for such products are easily mapped into computation and/or data retrieval at multiple locations. (This transparency is needed to enable sharing and optimization across diverse, distributed resources, and to keep application development manageable.)

As we shall see, transparency is an important aspect of our Data Grid architecture. In its most general form, transparent access can enable the definition and delivery of a potentially unlimited virtual space of data products derived from other data. In this virtual space, requests can be satisfied via direct retrieval of materialized products and/or computation, with local and global resource management, policy, and security constraints determining the strategy used. The concept of *virtual data* recognizes that all except irreproducible raw measured data (or computed data that cannot easily be recomputed) need to 'exist' physically only as the specification for how they may be derived. The grid may materialize zero, one, or many replicas of derivable data depending on probable demand and the relative costs of computation, storage, and transport. In high-energy physics today, we estimate that over 90% of data access is to derived data.

We note that virtual data as defined here integrates familiar concepts of data replication with the new concept of data generated "on-the-fly" ("derived") in response to user requests. The latter is a research problem while the former is common practice. Furthermore, the two concepts are logically distinct: it can be useful and important to replicate data even if derived data is not being generated automatically. However, we believe that there are likely to be advantages to an integrated treatment of the two concepts. For example, an integrated treatment will make it possible for users to choose to regenerate derived data locally even if the data is already materialized at a remote location, if data regeneration is faster than data movement.

## 2.2 Numerical Requirements and Challenges

The preceding discussion emphasizes that Data Grid systems must be able to support a wide range of environments and applications. Nevertheless, it is useful to characterize in approximate terms the scale of the problems to be addressed, in order to identify areas in which we face the most significant challenges. Table 1, taken from the GriPhyN proposal, represents one attempt to summarize these requirements.

**Table 1: Characteristics of the physics experiments targeted by GriPhyN and PPDG**

| Appli-cation | First Data | Data Rate MB/s | Total? Raw? Data Volume (TB/yr) | User Comm-unity | Data Access Pattern | Compute Rate (Gflops) | Type of data |
|---|---|---|---|---|---|---|---|
| SDSS | 1999 | 8 | 10 | 100s | Object access and streaming | 1 to 50 | Catalogs, image files |
| LIGO | 2002 | 10 | 250 | 100s | Random, 100 MB streaming | 50 to 10,000 | Multiple channel time series, Fourier transformations |
| BaBar | 2000 | 15 | 500 | 500 | Object Database | TBD | Raw and processed events (?/event) need official numbers |
| D0 | 2001 | 15 | 350 | 500 | Sequential access to files | TBD | Raw and processed events (250Kbytes/event) |
| STAR | 2000 | TBD | TBD | TBD | TBD | TBD | TBD |
| JLAB | 2000 | TBD | TBD | TBD | TBD | TBD | TBD |
| ATLAS/ CMS | 2006 | 100 | 5000 | 1000s | Streaming, 1 MB object access | 120,000 | Events, 100 GB/sec simultaneous access |

The experiments taking data today—SDSS, STAR, BaBar and D0—have existing data analysis systems that must be extended to provide the distributed computation and data transport required in the above table. The Data Grid services being developed must be integrated carefully into these systems to maintain their continuous operability and performance.

The following are more detailed estimates of the scope and performance demands associated with an LHC experiment around 2005:

- 200 sites with significant computing and storage resources ("central" and "regional" centers).

- A total of 30,000 computers and 15 PB of storage.

- 5 PB of unique data.

- 50 million logical files

- 500 million physical files

- 20 million physical files catalogued at a (large) site

- Average query time for the replica location service of 10 milliseconds

- Maximum query time for the replica location service of less than 5 seconds

- Replica location service query rates of 10 to100 per second (burst)

- Replica location service query update/insertion rate of 5 to 20 per second (burst)

We can also point to application scenarios that could lead to significant increases in requirements in several areas. Need to point to the monarc documents here http://home.cern.ch/~barone/RC/RCA5.txt

http://monarc.web.cern.ch/MONARC/docs/monarc_docs/1999-02.html For example, the resource estimates for LHC experiments assume that we are only harnessing resources at a modest number of dedicated regional computing centers. Now it could be that with appropriate software, we could increase accessible computing power dramatically by harnessing idle workstations at all participating institutions. Changes in the event trigger rate can also cause significant changes in the resource requirements.

# 3   Data Grid Architecture Elements

The discussion above indicates some of the principal elements that we can expect to see in a Data Grid architecture. In this section, we start the process of translating this general discussion into specifics, first by presenting a workflow view of Data Grid operations and then by an analysis of the different types of services required in a system that supports this workflow.

## 3.1   Workflow View

Figure 1 presents one view of the major elements of a typical Data Grid end-to-end application. In this picture, the elements inside the large lower box are intended to be domain-independent, Data Grid services.

1. An *application* workflow specification invoked by a Data Grid user specifies a set of Data Grid operations that are to be performed. In the proposed architecture, application workflows are specified via an abstract Grid Directed Acyclic Graph (G-DAG) which provides a standard syntax and semantics for representing Grid operations. The G-DAG

representation is discussed in Section 6.2.  This request will, typically, be somewhat abstract, referring to data products without regard to their location and/or materialization.

2. The abstract G-DAG is passed to a *request planner* that translates the original request into a *concrete* G-DAG representing a more detailed plan specifying data movement operations and computations.  In the course of developing this G-DAG, the request planner may consult catalogs, information services, and so forth.  We note that the degree to which this G-DAG specifies concrete actions vs. leaving decisions to execution is a topic of debate; our intention is to define a framework in which various alternative strategies can be investigated. The request planner needs to interface with the user/application for example to provide information about the estimated duration of the plan execution. It is possible, that the estimated execution time of the amount of resources needed to execute the plan are too large for a given user/application.

3. The G-DAG is passed to a *request executor* that manages the execution of the request on the Data Grid, potentially locating resources, mapping operations to resources, tracking progress of operations, handling various failure conditions, and so forth.  As computation proceeds, the request executor may also update various catalogs, for example to record that additional data products have been materialized and/or replicas produced, and to document the procedures followed to produce new data products.

4. The request executor may call upon other services, for example a *reliable data transfer service* to orchestrate data movement and/or create additional data replicas and *compute services* to perform computation.

In addition, various system components operate:

- *Accounting* systems keep track of resource usage, making it available for purposes of reporting, tracking, policy enforcement, and so forth.

- *Monitoring* systems perform active monitoring for purposes, for example, of intrusion and anomaly detection.

- *Replica creation* systems monitor system behavior with the goal of identifying when and where replicas should be created (or deleted), and generate job requests appropriately.

It is possible that the mechanisms in place for the request executor will be insufficient to deal with failures, for example if the derived data assumed to exist is not found. In such cases, the request executor needs to interact with the request planner to generate an alternate execution scenario.

In support of these components, we have a variety of services and resources:

- *Data catalogs* maintain information about data itself (metadata), the transformations required to generate derived data (if employed), and the physical location of that data.

- *Community authorization services* maintain information about the policies that govern who can use what resources for what purposes, and enforce these policies.

- The various *resources* that we must deal with: storage systems, computers, networks, catalogs, and code repositories.  (In the rest of this text, we use the word "resource" to refer to this range of entities, unless otherwise noted.).  These resources may include generic Grid resources such as compute and storage servers, or application specific resources and services such as database servers.

- An *information service* supports the discovery and monitoring of Grid resources as well as application specific services and resources.

Notice the clean separation of concerns expressed in Figure 1 between virtual data (catalogs) and the control elements that operate on that data (request manager). This separation represents an important architectural design decision that adds significantly to the flexibility of a Data Grid system.

In addition, we of course have the diverse and complex software systems constructed by partner science projects to perform their data collection and analysis. While this software is represented in Figure 1 as a small box, in most applications (e.g., those found in high energy physics) considerable development may well be required at this level in order to construct application systems that can exploit Data Grid mechanisms effectively. For example, we can have sophisticated application-specific *request formulation tools* that enable the end user to define data requests, translating from domain-specific forms to standard request formats, perhaps consulting application-specific ontologies. However, this machinery is beyond the scope of this document. (Unless someone wants to volunteer to develop requirements and specifications.)
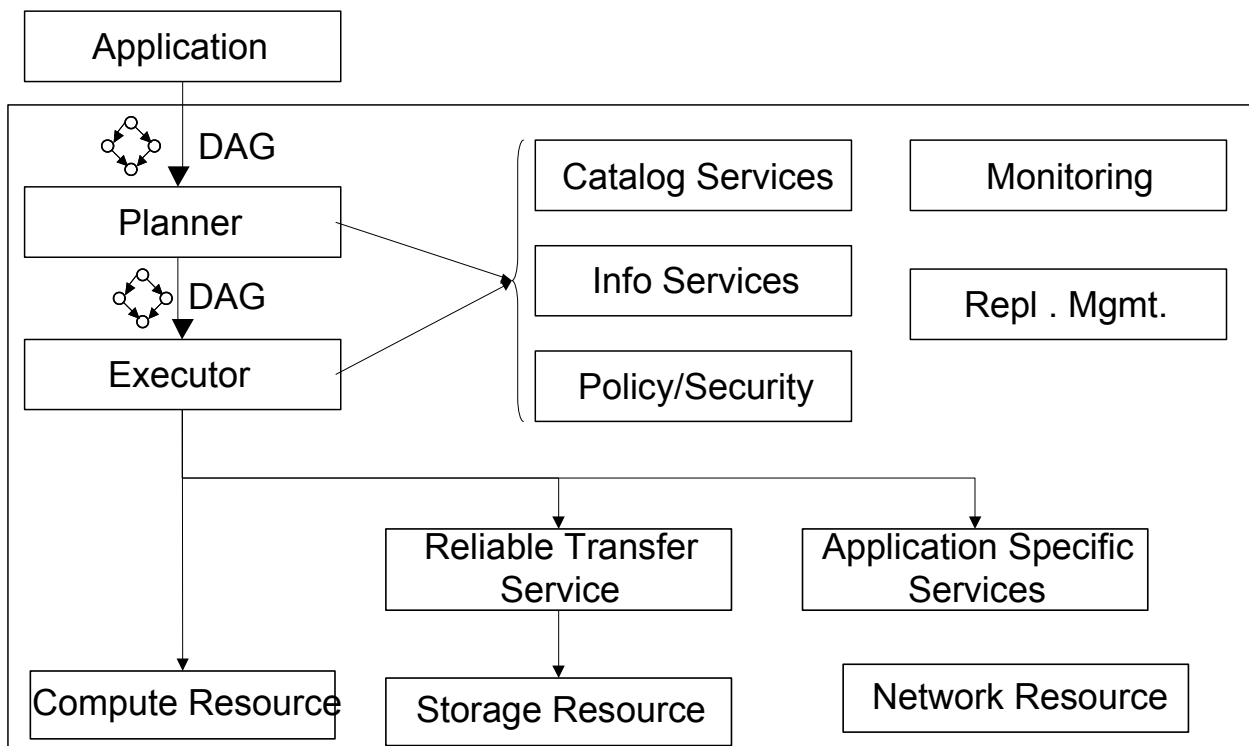


**Figure 1: Schematic showing the principal elements of the Data Grid reference architecture ITF: What else do we need if we want to be comprehensive? E.g., we don't show code repositories. .**

The architecture depicted in Figure 1 provides the flexibility needed to enable data management as envisioned by the experiments. For example, in CMS, a job is described as a set of possibly many subjobs (each being one executable) (http://kholtman.home.cern.ch/kholtman/griphynaug23.ppt [there might be a better reference, such as a doc]). The subjob descriptions are location independent. The data flow between subjobs is represented in terms of file sets, which are specified in terms of logical file names. Error rules are also provided in the subjob descriptions. The grid scheduler optimizes job execution by mapping subjobs to sites, generating any necessary data replication instructions.

Grid-wide execution service can use error recovery rules to recover automatically from common classes of failure.

## 3.2   Architecture Overview

Experience with a range of Grid applications and architectures tells us that the application-specific view of Data Grid architecture does not tell the whole story.  When accessing storage or compute resources, we require resource discovery and security mechanisms. When developing request plans, we need to be able to capture and manipulate such plans.  We also require logic for moving data reliably from place to place, for scheduling sets of computational and data movement operations, and for monitoring the entire system for faults and for responding to those faults.  These various mechanisms and services are for the most part application-independent and can be developed as independent mechanisms and reused in different contexts.

**Figure 2: Schematic of Grid architecture**

These observations lead us to view Data Grid mechanisms as being not independent of, but rather built on top of, a general Grid infrastructure [14].  Thus, a Data Grid system is defined, first of all, by a set of *basic Grid protocols* used for data movement, name resolution, authentication, authorization, resource discovery, resource management, and the like.  These basic protocols underpin all other Grid services and, as discussed in [14], are the basis for interoperability between Data Grid systems and with other Grid deployments.  We adopt the protocols defined by the Globus Toolkit as they represent the de facto standard for Grid systems.  In terms of Figure 2 (from [14]) these protocols comprise the Connectivity and Resource layer.

Building on this base, a Data Grid provides the programmer with three general classes of component:

- We have the *resource services* that provide Data Grid applications with access to individual Grid resources, whether computing, storage, network, or other physical devices.  These services are defined by the protocols used to invoke them and by the behaviors that they exhibit in response to protocol messages.  As we shall explain, Data Grids can exploit standard protocols but can benefit from specialized behaviors in the Fabric elements to which these services provide access.  (These are the services provided by the Resource layer in Figure 2.)

- We have the higher-level *collective services* that support the management and coordinated use of multiple resources.   Here, Data Grids introduce specialized requirements, in such areas as catalog services, replica management services, community policy services, coherency control mechanisms for replicated data, request formulation and management functions for defining, planning and executing virtual data requests, and replica selection mechanisms.  We can also reuse standard collective services, for example, for resource discovery.  (This is the Collective layer of Figure 2.)

- Finally, we have the various *programming tools* used to construct applications, and of course the applications themselves.

We structure the discussion that follows in terms of these categories.

Specific applications exploit Data Grid capabilities by invoking services at the Collective or Resource level, typically via supplied APIs and SDKs that speak the protocols used to access
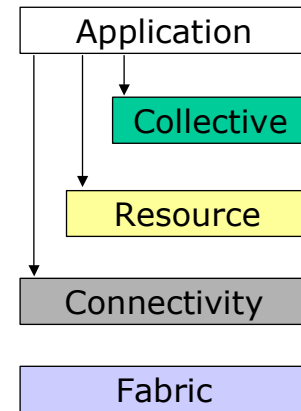
those services.  To be successful, our architecture must make it feasible to adapt this experiment-specific software for Grid operation not by rewriting it but rather by layering it on top of the services and APIs provided by the elements just described (request manager, catalogs, Grid services, etc.).  Reconciling the demands of the experiments with the capabilities provided by Data Grid software is an ongoing process of negotiation and experiment.
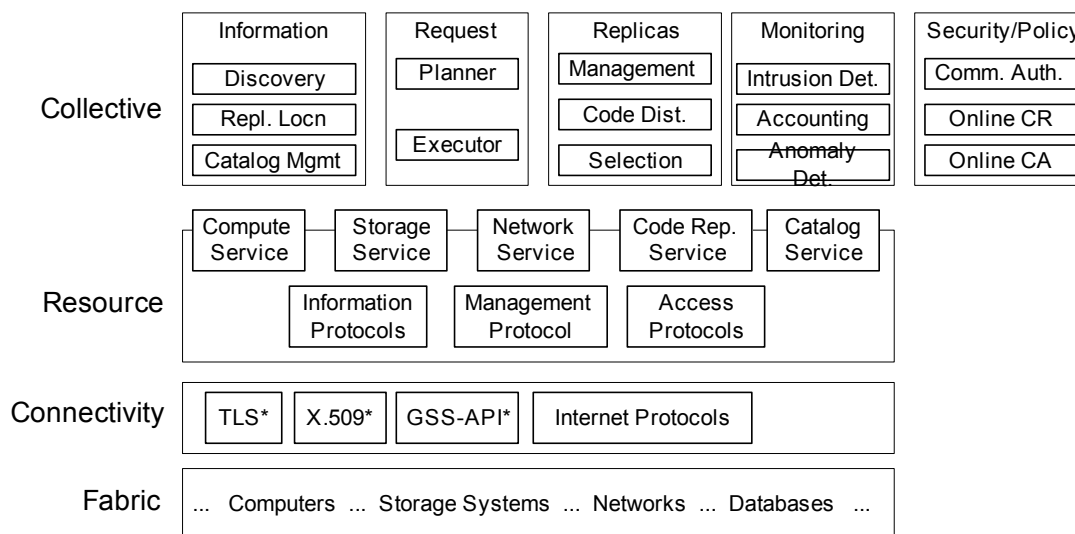


Figure 3: A far-from-complete listing of major Data Grid Reference Architecture elements, showing how they relate to other Grid services. Shading indicates some of the elements that must be developed specifically to support Data Grids.

# 4   Basic Grid Protocols

We assume the existence of general-purpose protocols for communication, authentication, resource discovery, and resource management.  We describe these protocols briefly here as background for the discussion that follows.  While these elements are invisible to the Data Grid user, they are important from the perspective of Data Grid developers, as they define the "neck" in the protocol hourglass, below which can be placed many different resources and above which can be implemented many services.  For example, standard authentication mechanisms allow different Data Grid developers to develop different components secure in the knowledge that there are standard mechanisms for establishing the identity of users and resources.  Similarly, a standard access protocol for storage elements allows developers of storage systems and developers of higher-level capabilities (e.g., replica management) to work independently.

## 4.1   Communication

Data Grid applications require communication mechanisms that can, depending on context, provide for secure, high-speed, and/or performance-guaranteed transmission.  The basic Internet protocols have deficiencies in each of these areas, but we can nevertheless work with them while waiting for improved capabilities.  Hence, our strategy in this area is to use basic Internet protocols for communication, name resolution, and the like.  As advanced capabilities such as IPv6, IPsec, and quality of service (QoS) become generally available, we should investigate their utility for Data Grid applications.

Current status: Considerable experimentation with QoS has been conducted (e.g., [15]), but lack of broad deployment has so far prohibited any practical application.

Future plans: Integrate IPv6 support into Grid communication libraries. Experiment with wavelength allocation in WDM networks.

## 4.2   Authentication and Authorization

Data Grid requirements for authentication and authorization include:

- Secure, bilateral authentication of users and resources: i.e., mechanisms that allow two entities (e.g., user+service, user+resource, service+resource) to verify each other's identity.

- Confidentiality of data transferred.

- Access control (authorization) expressible by resource, code, and data owners. This corresponds to enforcement of local policy. Examples of such local policy may be file-system and CPU usage quotas, usage priorities, ACLs, etc.

Other requirements relating to community authorization and the publication and discovery of access control policies are discussed later.

We address these requirements by adopting the public key infrastructure (PKI)-based Grid Security Infrastructure (GSI) [13], which provides the following essential capabilities:

- *Single sign on*, allowing a user to authenticate once and then delegate rights to a computation that runs on their behalf, accessing resources without further requiring the user to engage in further authentication operations.

- *Mutual authentication* of computations and resources.

- *Authorization* via call outs to local policy.

- *Message privacy and integrity*, if required.

These services are accessible via a standard API (GSS-API), from the GlobusIO package for secure socket-based communication, or from the many tools that have integrated GSI support. Other supporting tools address credential management.

Current status: GSI is broadly adopted and GSI-enabled versions of a wide variety of tools exist, including FTP servers and clients, Secure Shell, Condor [19], SRB, etc. Discussions of standard Certificate Authority (CA) policies are ongoing in GGF.

Technical details: See http://www.globus.org/security/ for details on GSI.

Future plans: An R&D project funded by DOE's SciDAC program is continuing to evolve GSI, addressing issues of usability and scalability (credential management), restricted delegation, etc. Reach agreement across Data Grid and other Grid projects (e.g., DTF) on standard CA policies. Address issues of community policy, as discussed in Section 8.1 below.

## 4.3   Resource Discovery and Monitoring

We next require mechanisms that support the discovery, characterization, and monitoring of Grid resources. We adopt those defined and implemented by the Globus Toolkit's Meta Directory Service (MDS-2).

- A registration protocol, the Grid Resource Registration Protocol (GRRP), that allows a resource or service (yet more generically, a "sensor") to notify another entity of its availability and how to contact it for purposes of enquiry or control. Such a protocol

might be used, for example, to register resources with an information service that then constructs an index of known storage systems.

- An inquiry protocol, the Grid Resource Inquiry Protocol (GRIP), used to enquire as to the structure and state of a resource or service.  A data model and query language are also defined.  For example, in the case of a storage system, an enquiry protocol might allow an application to query a storage system concerning its capacity, speed, availability, and functionality.

In brief, we require that all resources and services in a Data Grid speak these protocols, hence making it possible for us to discover them and determine their properties.  More details on this technology can be found in [8] (and in [17] for GRRP) and at www.globus.org/mds.

Given the basic protocols a wide variety of different information organization and distribution schemes can be created via the construction of information indexes.  Indexes may be queried via Grid Information Protocols (GRIP) or make support specialized query protocols such as ODBC.

Current status: MDS-2.1 defines data models and provides implementations of sensors for computers, storage systems and several other devices.  An interface to the Network Weather Service (NWS) [31] provides network performance information. Recently, sensors have been developed for GridFTP servers, that provide information on transfer performance.  GSI security is supported for accessing MDS-2.1 components and information provided by them..  Simple directory services are available.

Technical details: See www.globus.org/mds for more information.

Future plans: We need to: expand greatly the set of sensors supported, to include, for example, sensors for catalogs.  Hand in hand with the development of additional sensors, we need to define information schemas to facilitate the publication and interchange of data generated by these sensors.  With regard to the basic protocols, they need to be extended to  provide push-mode as well as pull-mode inquiry.  We also plan to look at mapping the information protocols onto different transport mechanisms such as SOAP.  We will develop more specialized information indexes with an initial focus on "archiving indexes" and indexes optimized for a variety of query types, including relational queries.  Finally, we need to  develop access control further to support CAS.

### 4.4   Resource Management

Resource management interfaces are need to reserve, allocate, and manage the underlying resources that are accessible to the data grid.  The different types of data-grid resources are described in the following section.  We adopt the Grid Resource Allocation Management (GRAM) protocol as the method for communicating with resources.  GRAM defines protocol operations for resource allocation, as well as monitoring and controlling the allocated resource.  Resource requests are made via a standard resource specification language (RSL).  Current GRAM protocols focus on the management

Current status: GRAM 1.5 provides a reliable interface to computational resources.  C, Java, and Python API bindings exist.  GRAM 1.5 improves on previous GRAM implementations by providing more robust failure recovery mechanisms and additional RSL attributes, thanks to contributions from the U.Wisconsin Condor group.  Limited support for advanced reservation and for additional resource types exists in prototype form [15, 24, 25].

Technical details:  See www.globus.org/gram for more information.

Future plans: A new version of the GRAM protocol (GRAM 2.0) is being designed.  The focus of this protocol revision is to provide greatly increased robustness and scalability through the use of

soft-state protocols and hierarchical resource management contexts. Additional features will include the ability to specify and enforce a wide range of delegated resource management policies, and support a variety of flexibly resource specification notations.

# 5 Data Grid Resources

We now define the primary resources with which we are concerned in a Data Grid environment. We consider five classes of resources: storage systems, compute systems, networks, code repositories, and catalogs.

For each, we define the protocols used to access them and the behaviors that we expect these resources to provide. In general, each resource may speak three general classes of protocol:

- *Information protocols*, used to notify the data grid environment of its existence and to allow others to determine its status and configuration;

- *Management protocols*, used to control the behavior of the resource, for example by making reservations; and

- *Access protocols*, used to initiate and control resource use.

Current status: As described above, the Globus toolkit defines GRRP and GRIP as the basic protocols for providing information. In the current toolkit implementation, resource management protocols are restricted to prototype implementations (i.e. GARA), while resource access is supported via GRAM (for computing) and GridFTP (for data).

Technical details: As discussed above, see www.globus.org/mds for information about information protocols and www.globus.org/gram for details on management and access protocols.

Future plans: The GRAM 2.0 protocol will support integrated management and access of resources within the context of a single protocol framework. In addition, information and resource protocols will most likely migrate towards a common, soft-state notification protocol derived from the ideas found in GRRP.

## *5.1 Storage Systems*

We define a storage system not as a physical device but as an abstract entity whose behavior can be defined in terms of its ability to store and retrieve named entities called files. Various instantiations of a storage service differ according to the range of requests that they support (e.g., one might support reservation mechanisms, another might not) and according to their internal behaviors (e.g., a dumb storage system might simply put files on the first available disk, while a smart storage system might use RAID techniques and stripe files for high performance on observed access patterns). They can differ according to their physical architecture (e.g., disk farm vs. hierarchical storage system) and expected role in a Data Grid (e.g., fast temporary online storage vs. archival storage). They can also differ according to the local policies that govern who are allowed to use them and their local space allocation policies. For example, an "exclusive" storage service might guarantee that files are created or deleted, and space reserved, only as a result of external requests; while in the case of a "shared" resource these guarantees might not hold. A related issue might be whether a storage system guarantees that files are not deleted while in use.

We believe that within the context of a Data Grid architecture, it is important to reach consensus on a small set of "standard" storage system behaviors, with each standard behavior being characterized by a set of required and optional capabilities.

Regardless of the type of storage system, we can establish base line capabilities that any storage system should have. These required capabilities are:

- Store, retrieve, and delete named sequences of bytes (i.e., files). Names may be hierarchal, as in the case of typical disk based storage systems, or flat, as is sometimes found in archival storage systems that utilize tape volume identifiers to identify files.

- Provide mechanisms for controlling what operations can be performed on files and the storage system (access control)

- Report on basic characteristics of the storage system, such as total storage capacity and available space.

Additionally, a storage system may support a number of basic optional capabilities including:

- Reserve space for a file.

- Reserve disk bandwidth (guaranteed transfer rate).

- Support a variety of management operations including initiation of high performance (i.e. parallel) transfer, explicit caching or staging of data, etc.

- Ability to report on more detailed attributes such as supported transfer rate, latency for file access (on average or for a specific file), etc.

- Co-located compute resources (i.e., this storage element may support "local" access mechanisms, such as Unix read/write operations).

In addition to the basic functional aspects of storage systems, the *policy* under which a storage system is operated will have a significant impact on how it can be used. One policy element typically found in storage systems concerns the amount of storage that may be used by any one user or application. Specifically, many storage systems support the use of fine-grained quotas (i.e., available space may be larger then the largest file a specific user may be able to store). Storage systems can also be distinguished by the policy that they enforce with respect to the assurances they make with respect to the *time duration* for which they will store files. Specifically, we can identify the following of storage policies:

Finite storage duration. In this situation local operational policy does not guarantee longevity of data. Scratch storage and network based storage element (data caches) fall into this category. In storage systems that implement this policy, additional optional behaviors include:

- Guarantee that files are not deleted while a transfer is in progress.

- Accept hints about desired data lifetimes (e.g., pinning)

Arbitrary storage duration: At the other end of the spectrum are storage systems that provide open-ended storage of files. In its most general formulation, this can result in a storage system with potentially unbounded requirements for storage capacity. To manage the costs associated with building such a system, frequently, several different types of physical storage (such as disk and tape) are integrated into a single system and files are transparently migrated from one to the other based on requests. On such hierarchal mass storage systems, specialized optional behaviors may include:

- Request staging to reduce latency for access to specific files

- Guarantee that files are not migrated while a transfer is in progress

- Accept hints about usage patterns to aid in managing higher levels of the storage hierarchy

- Accept reservations on various levels of the storage hierarchy.

Note that if explicit migration between physical storage types is supported, the storage system can be modeled as a collection of separate storage elements, each with its own management policy and performance characteristics.

By combining different storage systems and policies, a variety of different higher level storage services can be created. For example the hierarchical storage resource manager operations [Shoshani, 1998 #1553] can be defined in terms of a set of storage systems, some representing tape archives (operated under arbitrary storage duration policies) and associated sets of "cache" storage systems, operated under finite storage duration policy. HRM protocol operations such as those proposed in [Bird, 2001 #1758] can then be implemented in terms of basic storage system protocols as applied to the well defined configuration of elemental storage systems and policies.

Current status: GridFTP is available as a standard storage interface, and MDS interfaces to storage systems have been developed, although they require further development. At LBNL, work continues on development of a Storage Resource Manager (SRM) [26] that can provide additional QoS guarantees to clients.

Technical details: www.globus.org/datagrid has more information on GridFTP.

Future plans: Develop standard object classes for characterizing storage systems in terms such as those listed above. Develop storage systems that support reservations. Develop layered protocol specifications for providing integrated support for SRM functionality on top of basic storage system services.

## 5.2   Compute System

We define a compute system not as a physical device but as an abstract entity whose behavior can be defined in terms of its ability to execute programs. Various instantiations of a compute service will vary along such dimensions as their aggregate capabilities (CPU, memory, internal and external networking) and the quality of service guarantees that they are able to provide (e.g., batch scheduled, opportunistic, reservation, failover, co-allocation of multiple resources). They may also vary in their underlying architecture (uniprocessor, supercomputer, cluster, Condor pool, Internet computing system), although as with a storage service, these details should be of less importance to users than the behaviors that are supported.

As with storage systems, we are interested in defining standard behaviors. Here are our candidates:

Compute farm: On-demand access to computing for single-processor jobs, with or without reservation.

Parallel computer: On-demand access to computing for multi-processor jobs, with or without reservation.

Current status: We use the Globus Toolkit's GRAM protocol and API to obtain access to remote compute systems, with extensions developed by the Wisconsin Condor group providing enhanced reliability.

Future plans: Develop GRAM-2 to provide enhanced reliability and scalability. Integrate dynamic account allocation mechanisms as a means of providing protection without requiring pre-existing accounts for all users.

### *5.3   Network*

Networks are a critical Data Grid resource, and the ability to discover, understand, predict, and (ideally) manage their performance is critical to our ability to meet application performance goals.  In more detail:

- We need to be able to determine the capacity of a network and determine its current utilization.  Real-time monitoring is required for this purpose.

- We need to be able to detect and diagnose performance problems.  While this requirement arises for any Data Grid component, the distributed and often "black box" nature of networks makes it particularly important.

- We would like to be able to predict expected future utilization.

- We would like to be able to manage the allocation of bandwidth to different purposes.

Each of these requirements is important, but only the first two seem to be on the critical path for Data Grid deployment.

Current status: We use Network Weather Service (NWS) [31] to measure network performance and provide some short-term predictions.  This is integrated into MDS.  Various tools exist for determining trends and diagnosing performance problems: e.g., Pinger and Pchar.  Quality of service mechanisms are addressed by the GARA bandwidth broker and advance reservation system [15], but are not supported in today's networks.

Technical details: Information about Network Weather Service can be found at http://nws.cs.utk.edu.  .

Future plans:  Support for managing network bandwidth explicitly as a resource is one of the design goals for the GRAM 2.0 protocol.

### *5.4   Code Repositories*

A code repository is a storage service used to maintain programs, in source and/or executable form.  We discuss it as a distinct entity because of the critical role that online access to programs plays in Data Grids and because code repositories introduce some specialized requirements.

Like a storage or computational resource, a code repository should provide information about its contents, as well as providing access to the actual code elements.  The capabilities require in a code repositories remain to be defined, but should presumably include the following:

- Ability to manage multiple revisions of the same application as well as versions for different architectures.

- Provide information about code, including complete, platform requirements for execution, performance profiles, and a description of input and output parameters.

- Support for dynamic linking and shared libraries.

- Access control and policy enforcement.

As with compute and storage systems, there are many instantiations of this basic concept (e.g., many physics applications have developed their own code repositories).

Code distribution is a related requirement; we discuss this in Section 9.2.

Current status: The GridCVS utility supports GSI-authenticated access to CVS repositories.  Within the Atlas experiment, the PACMAN system

(http://www.usatlas.bnl.gov/computing/software/pacman/) is being used for distribution of precompiled software packages (either tarballs or Linux RPMs).

Technical details: See www.globus.org/gridcvs.

Future plans: We need to collect comprehensive requirements for code repository and distribution services, and either import this capability from elsewhere or develop it ourselves. We also need to consider how to collect, represent, and store performance information used for query estimation. We need to specify the relationship between the Transformation Catalog (TC) [ref], which provides information a mapping between a transformation and its location. TC can also include metadata about the transformation, such as who has designed it, who validated it etc….

## 5.5  Catalog Services

A catalog is a storage service used to maintain mappings of some sort. We discuss it as a distinct resource because of the critical role that online access to such mappings plays in Data Grids and because catalogs introduce some specialized requirements (see [Holtman, 2001 #1759], for example).

In [9], we identify a need for the following catalog services:

- *Metadata catalog* [4] that handles mappings from "entity id" to "entity attributes" for the entities of interest, which are files in the short term but may be objects in the future.

- A *Grid Container Management Service* (GCMS) will be used to associate objects to logical containers as well as to provide the ability to import/export data into the grid.

- *Replica catalog* [2] (or, as we argue in [5], *replication location service*) that maintains information about the physical location of replicas of files (or containers in the future).

- Other catalogs concerned with supporting transparency with respect to materialization, such as derived data catalogs, meta derived data catalogs, and transformation catalogs [9].

Notice the clear distinction made between metadata and replica catalogs. This distinction is important for three reasons. First, many applications maintain their own metadata catalogs; we do not want to force them to change them to support replica information. Second, we anticipate access patterns on metadata and replica data to be different. Third, as we discuss in more detail in Section 7.3, it is not necessarily vital that replica location information always be completely consistent—while metadata typically must be.

All of these catalogs require appropriate support for access control and policy enforcement. All can have challenging scalability requirements in some Data Grid environments: e.g., LHC experiments talk about 10s of millions of files and 100s of replica sites. In addition, we note that like storage and computational resources, catalogs should provide information about their availability and contents via MDS, as well as providing access to the actual data elements. We discussed this requirement in Section 4.3 also.

In the rest of this section, we discuss requirements for metadata

### 5.5.1  Metadata Catalog

A metadata catalog maintains a mapping from entity name (e.g., object name or logical file name) to entity attributes. These attributes are an essential part of the entity, and must be maintained and protected with as much care as the data. (In fact, in some cases, metadata may be *more* important: we may be to recreate data but not metadata.) We are aware of the following potential requirements:

- *Security*: access control for reading and writing. May be on a per-attribute basis (?). Integration with a community authorization service will be important.

- *Scalability*: In some domains, the number of entities can be large (certainly tens of millions, perhaps significantly more). We have been told that read and write rates will be low (tens or at most hundreds per second), but we need more information on this point. However, it is expected that reads will dominate the accesses.

- *Persistence*: Metadata needs to be protected against unplanned deletion. To what extent remains unclear.

- *Fault tolerance*: Mirroring of metadata may be important, to ensure accessibility in the event of network partitions or other failures.

- *Extensibility*: We may need to record attributes relating to Data Grid operation, e.g., concerning the location of a "master" copy of a file, the transformations used to generate a piece of derived data, and so forth.

All the above requirements are also important to the DMDC [ref], which provides similar functionality to the MDC, but in the virtual space.

We do not yet have good numerical estimates of performance requirements.

Current status: A variety of implementation options are possible and we have not decided on (indeed may not be able to) decide on a single approach. Metadata catalogs are typically provided by applications, although with additional attributes required for Data Grid operation [9]. In this case, we may simply be concerned with providing for GSI-enabled remote access. It would also be desirable to provide a "standard" metadata solution for use when no other solution is available, although it is not yet clear what that standard should be. SRB [4] uses an SQL-like syntax. The Open Archives Initiative Protocol [27] may be relevant. Ultimately we need to address issues of persistence, fault tolerance, and scalability.

Future plans: Define requirements for; develop; and make available a standard metadata solution.

### 5.5.2   Replica Catalog

Requirements and current status of work on replica location services are discussed in Section 7.3.

Current Status: we have a design and implementation of a centralized replica catalog, which maps logical collections and logical file names to a particular physical file instance [Allcock, 2001 #1662].

Future plans: a new design is being proposed for enabling a construction of a distributed replica location service.

### 5.5.3   Transformation Catalog

The transformation catalog will be used to store information about the transformations, which need to be used in order to process data as requested by the user. The catalog can be used to process raw or derived data products. Data derived using the transformations will be possibly placed in the Metadata catalog or the Replica Catalog.

A major factor in deciding on the design is the need for information about the architecture and the platform of the machine where the computation is to be done. Another important factor is whether a binary is available or not for that architecture. If binaries are available the shared libraries are assumed to be included in the tarball distribution as they may be required to run the executable. We also need to provide compiler information (which compiler was used to generate the code) as well as the flags used. If the source is distributed we need to provide a means for describing how

to build the software. Factors such as cost of running or accessing the transformation as well as the minimum resource requirements have to be taken into consideration.

Current Status: Several prototype transformation catalogs have been designed and are being evaluated within the context of CMS and LIGO experiments.

Future Plan: Discuss the current design and understand the relationship to the code repositories.

### 5.5.4   Virtual Data Catalogs

Work is under way to develop requirements for, and explore via prototypes, the virtual data management catalogs, such as the DMDC and the DDC [Deelman, 2001 #1663]The derived data catalog (DDC) records information associated with derived datasets, such as the transformation needed to generate the derived dataset, the cost of that transformation, and a unique derived dataset name.  DDC deals with derived datasets at the level of data objects. "Meta attributes" of the catalog entries derived data, such as owner, name, author, and creation-time are also recorded here.  This element together with the DMDC (below) and the transformation catalog provide information required to support the manipulation of derived data.

The derived metadata catalog (DMDC) is similar to the MDC but it maps from a set of application-specific attributes to a derived data object id(s) that indexes into the DDC.

The difficulty is in the understanding of how to name virtual data products and how do describe the materialization process. One idea is to integrate the G-DAG ideas and structures described in Section 3. For example, in the DDC, we might want to describe the process of obtaining the virtual data using a G-DAG description.

### 5.5.5   Other Catalogs

Work is under way to develop requirements for, and explore via prototypes, the other catalogs referred to above.  More detail will be provided in a subsequent version of this document.

Current status: There is a document [Deelman, 2001 #1663] that specifies the structure of the virtual data catalogs.

Future plans:  Evaluate the design of the catalogs using virtual data as described by the experiments.

# 6   Request Planning and Execution Services

The first higher-level services that we discuss are those concerned with request planning and execution.  From the perspective of our Data Grid architecture, we must:

1. Define or use established protocols for the request management services to interact with the catalog and information services

2. Define a common representation for the (typically partially ordered) sets of tasks or operations that must be performed on the Data Grid, and

3. Define a protocol for submitting task descriptions using the above representation to planner and execution services.  We must also define protocol messages for monitoring and controlling progress as these services operate on the submitted tasks.

4. Define protocols and policies for the execution services to operate in case of failure.

Many different representations of task sets exist and could be considered for our purposes. However, we propose to represent data grid tasks via a graphical data-flow-type notation that models the operations (nodes) and the flow of data (arcs) and that defines one or more triggering

rules (all available, N out of M available, etc).  Data-flow concepts have proven to be useful in a variety of workflow, scripting, and composition applications (cite AVS, workflow) and we believe that they are powerful enough to represent the types of operational activities that need to be performed in the Data Grid context.

## 6.1   Abstract and Concrete Representations

Before a set of operations can be executed on a data grid, it must be completely instantiated: storage systems and physical file name specified, specific computational resources identified, specific versions and implementations of applications located, etc.  We refer to a plan that has this level of detail to be a *concrete* plan.

On the other hand, a task graph specified by a user might be expressed in terms of high-level, abstract operations, specifying, for example, names of programs and not specific versions, logical file names, data attributes, etc.  We refer to such a plan as *abstract*.

End users will tend to specify abstract plans.  Virtual data requests by definition will be abstract. Execution services will expect to see concrete plans.  At various points in the process of manipulating a plan, it may contain a mixture of concrete and abstract elements.  The planning and execution process will perform various activities designed to translate abstract plans into concrete plans.

## 6.2   Grid Directed Acyclic Graphs

Many different data-flow representations have been proposed: each with a different set of basic nodes and associated semantics.  Nevertheless, because they can be mapped into an underlying graph model, it is possible to create a common graph-based syntax to represent these different semantics.

While we admit the existence of alternative representations of task sets, we propose to use a specific simple representation for the initial version of the Data Grid architecture.  Specifically, we will represent data grid computations as Grid directed acyclic graphs (G-DAGs).

To discuss: how does this relate to various JCLs that are being discussed, e.g. in PPDG?

### 6.2.1   G-DAG Syntax

A directed acyclic graph (DAG) comprises a set of nodes connected by directed edges.  Both nodes and edges may be labeled.  We can represent a DAG graphically or in XML, as shown in Figure X.  (See Appendix X for more details.)  I propose that we define an XML syntax, I think that is the right thing to do.  But let's look at what XML tools already exist first.  What about choreography tools?  We should also define Miron's syntax



**Figure 4: From left to right, graphical, Condor, and XML representations of a directed acyclic graph with four nodes.**

## 6.2.1.1 DAGMan syntax

As a starting point we propose that the syntax used by the Condor system's Directed Acyclic Graph Manager (DAGMan). DAGMan is a meta-scheduler that sequences the submission of activities to Grid resources, submits jobs in an order represented by a DAG and processeing the results.

A DAG description consist of:

1. A list of the activities (i.e. programs) in the DAG.

2. Pre/post processing that takes place before/after the submission of any programs in the DAG to a Grid resource.

3. Description of the dependencies in the DAG.

The following is an example of a DAG that implements the diamond graph shown above (http://www.cs.wisc.edu/condor/manual/v6.2/2_10Inter_job_Dependencies.html):

```
Job  A  A.condor
Job  B  B.condor
Job  C  C.condor
Job  D  D.condor
Script PRE  A top_pre.csh
Script POST C mid_post.perl $JOB $RETURN
PARENT A CHILD B C
PARENT B C CHILD D
```

The first 4 lines specify the jobs and the scripts that contain the job specification. Additionally, the file specifies a pre-processing for job A and a post-processing for job C (next two lines). Finally, the dependencies between the task as shown in Figure 4 are specified in the last two lines.

### 6.2.2  G-DAG Semantics

We interpret a DAG as follows.  Nodes represent tasks to be performed.  Edges represent dependencies between tasks and are typically labeled with the name(s) of dataset(s) produced by the predecessor task and consumed by the successor task.  The semantics of a G-DAG are as follows: if two tasks are connected by an edge, then the predecessor must complete execution successfully before the successor can be executed.  Notice that this semantics prevents pipelining of tasks; we will need to see if it is too restrictive.

The proposed initial DAG semantics requires that if any task fails, then the entire G-DAG fails. If failure recovery is to be possible, these semantics require that each node in the G-DAG exhibit and all or nothing transactional behavior with respect to side effects.  Upon failure, it may be possible to resubmit the partially completed G-DAG (restart) or to restructure the G-DAG to use alternative resources (fail-over).  Note that in general, it would be desirable to augment G-DAG semantics to specify alternative execution paths as part of the execution plan.

As defined above, a plan and hence a G-DAG can be either an *abstract DAG* and a DAG that only includes references to physical files: a *concrete DAG*.  As illustrated in Figure 5, we can think of request planning operations as progressively rewriting an abstract DAG to an increasingly concrete form.  On the left, we see a request for a value V.  This has not been materialized, so the next step is to determine the computation required to produce it, hence producing the DAG in the center.  The final planning phase identifies the physical files and locations that will be used to perform the computation, as encoded in the DAG on the right.
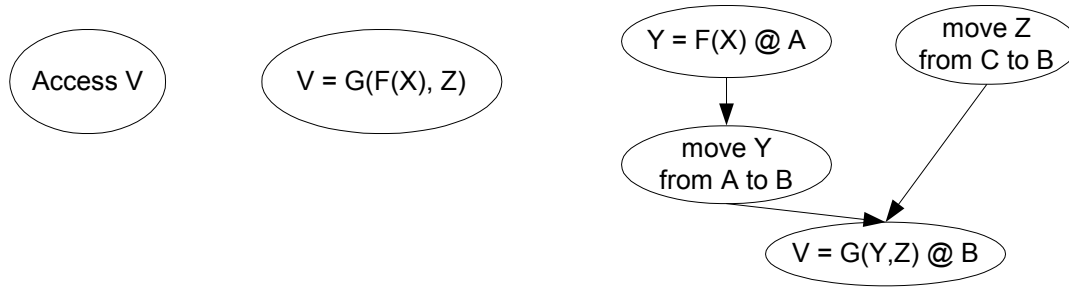
**Figure 5: Rewriting a G-DAG from, on the left, an abstract request (access a value V) to a computational request (center) and hence to a concrete set of computations and data movement operations (on the right). V, X, Y, and Z are data values, and A, B, and C are sites.**

The description of the request planning process in the preceding paragraph is likely to be overly simplistic, in that it does not allow for decisions made dynamically during execution. Dynamic decisions may be needed for several reasons. First, the data consumed by a computation may be determined only during its execution. Second, it may be desirable to delay decisions concerning where to execute tasks and/or which copies of data to access until execution time. Hence, in the terms of Figure 1, the DAG passed to the request planner may not be totally concrete. We anticipate that our G-DAG syntax and semantics will evolve over time as we learn more about these issues.

### 6.2.3   Planned Extensions

It is our intention to extend the G-DAG syntax and semantics in a number of areas, including the following:

- *Query estimation*: We want to allow nodes (and edges?) to be annotated with estimated costs, for use during the planning process.

- *Alternative plans*: We want to allow nodes (and edges?) to be annotated with information concerning what to do in the case of failure.

- *Client interface*: We want to allow the insertion of additional nodes, which can communicate with the client and provide information such as execution status.

These extensions are topics of current investigation.

Current Status.  We use the Condor DAGMAN toolkit [20] as our G-DAG reference implementation.  The system supports simple graph description in which one specifies the precedence relationship between named nodes, and the programs to be run at each node and local pre and post node processing.

Future plans:  While DAGMan provides a reasonable starting point for G-DAG notation and semantics, the discussion above illustrations a number of enhancements that we believe will need to be made.  These include:

- Create an XML based representation of DAG structure.  Investigate, for example, RDF [1], although this may be overkill.

- Extension of DAG structure to support additional attributions such as cost annotations and link annotations to support pipelining

- Extend DAG structure to accommodate error handling extensions and alternative execution paths.

- Investigate alternative dataflow structure such as those being considered for workflow applications.

## 6.3   Request Planning Services

Request planning issues are not especially well understood at this point in time, but are the topic of much current research in various application projects.  Our current understanding of the problem is that:

- The user provides a request as a G-DAG that refers to various data values without regard to their location or materialization.

- An entity called a *request planner* consults various catalogs and information sources to determine what data exists, and what computations and data movements may be required to process the request.  For example:

    o   If the data is materialized, the planner evaluates the costs of accessing the data from different locations. We might also evaluate the cost of re-computing the data, in case it is cheaper to do so.

    o   If the data is not materialized the various derived data catalogs are consulted to establish how to produce the data and the cost of doing so.

- Information services are consulted to establish resource availability and estimate the performance costs of data evaluation and retrieval.

- The output of this process (a plan or plans) is a more detailed G-DAG.

- Finally, it might be desirable to allow the user/application to decide whether, given the estimated cost, they want to proceed with data materialization or which way to proceed, batch vs. interactive for example.

Current status: We do not have any generic request planning capabilities.  However application-specific request planners for LIGO and CMS have been developed. In the LIGO prototype being currently developed, the user specifies the request in XML format (or via a GUI interface, which produces an XML request). The request planner identifies which data products need to be produced and design a plan in DAGMan's format.

Future plans: Develop request planning activities. Look into agent technologies as a possible framework. Although some of the planning is expected to be application-specific, we need to look at components, which are generic, such as finding the "best replica", "find appropriate compute/storage resources", etc.

## 6.4   Request Execution Services

The request execution services need to be able to take a detailed G-DAG description and execute the components on a specified resource. The components can represent data movement, computations, access to replica management services, etc….

The request execution services need to be able to be integrated with the request planner in order to deal with task execution failures. Consider the situation where the request planner has determined via the Replica Location Services (see Section 7.3) that a given file is available at locations X and Y. Using a replica selection algorithm, the planner decides to make a plan to access the data at location X. It is possible however, that by the time the request executor is accessing the data, the data is no longer present at location X. In that case, we have two choices:

1. the request planner can enough information for the request executor to be able to find another copy of the data

2.  interface the planner to the executor in such a way, that in the case of a node execution failure, the executor can ask the planner for an alternative plan.

Requirements: ???

Our initial execution service is based on DAGMan and Condor-G.  The syntax of a DAGMan plan was described above.  DAGMan supports very simple error semantics.  If any node in the DAG fails (as indicated by its return code), the entire DAG is aborted after all other independent nodes finish.  Instead of resubmitting the jobs, DAGMan generates a "Rescue DAG," that is functionally the same as the original DAG file, but it includes indication of successfully completed nodes. If the DAG is resubmitted, the jobs marked as completed will not be resubmitted. This Rescue DAG is automatically generated by DAGMan when a node within the DAG fails.

Request execution involves three distinct activities:

- *Request execution*: The needed resources are allocated (if possible) and the computation is executed. The Condor DAGMan represents the current state of the art in this area, providing required services such as credential refresh and reliable restart in the event of failure.  See [16] for some details.

- *Request monitoring*: We need to be able to monitor the progress of the request so that it is available to the user/application. The request execution service needs to be able to deal with failures and construct contingency plans. This functionality is currently part of DAGMAN. However, we need an interface from DAGMan to the application to make the desired monitoring information available.

- *Error recovery*: We need to able to recover from simple failures, such as unavailability of the data at a given location.

- *Information update*: Upon request completion, we need to notify and return the output to the user/application.  We also need to update catalogs to reflect that the data has been materialized.  Just how and when these updates occur is not fully understood.  For example, if a transformation produces a whole range of data values, we might not want to register them all, but instead discard them. To decide this issue we need to enhance our understanding of how the application programs (transformations in our model) behave.

Current status: The DAGMan system supports the reliable execution of concrete G-DAGs containing both computational and data movement tasks.  DAGMan is used in conjunction with Condor-G to provide the ability to refresh a users credential in the case of long running jobs and to supports access to GRAM- and GridFTP-mediated resources (i.e., computers and storage systems)

Future plans: Extend DAGMan to support more dynamic execution plans including plans with input/output sets that are determined at runtime, alternative execution paths and user-specified error recovery strategies.  Explore and implement catalog update procedures.  Define a network protocol to enable DAGMan to be run as a GSI-authenticated remote service. Define protocols and interactions between the request executor and the request planner and their relative responsibilities. Define the model for error recovery.

# 7  Information Services

We use the term information services to refer collectively to services concerned with resource discovery and monitoring.

## 7.1   Discovery and Monitoring Infrastructure

An information service supports enquiries concerning the structure, state, availability, etc., of multiple Grid resources.  Resource attributes may be relatively static (e.g., machine type, operating system version, number of processors) or dynamic (e.g., available disk space, available processors, network load, CPU and I/O load, rate of accomplishing work, queue status and progress-metrics for batch jobs).  Different information service structures may be used depending on the types of information to be queried and the types of queries to be supported.  For example, NetLogger [30] is frequently used for application monitoring.

Current status: We propose to adopt the Globus Toolkit's Meta Directory Service (MDS) information service architecture.  MDS uses the GRRP and LDAP registration and enquiry protocols to construct Grid Index Information Services (GIISs) that receive GRRP registration messages and maintain a list of active services; use LDAP queries to retrieve resource descriptions, from which they construct indices, with time to live information indicating how frequently indices should be updated; and use those indices to process incoming LDAP queries.

Future plans: Investigate further the GIIS functionality required in Data Grid applications.  Develop standard object classes for additional entities.  Address issues of archiving of monitoring information.

## 7.2   Collective Monitoring Services

The broad deployment of standard information access protocols such as those provided by MDS makes it feasible to construct a wide variety of monitoring systems that can perform various sorts of anomaly detection and system characterization.  However, little work has been done in this area to date.

*Intrusion detection*.  Techniques will clearly be required for monitoring the ensemble of Data Grid resources to detect coordinated attacks [10, 22].

*Anomaly detection*.  Techniques are required for monitoring the ensemble of sensors associated with a Data Grid to detect various types of anomalies, such as network performance problems, scheduling problems, and attacks.  Having detected an anomaly, such systems can also potentially attempt to correct detected problems.

*Resource characterization*.  Monitoring can also be used to develop summaries of resource characteristics, relating for example to relative quality and availability.

## 7.3   Replica Location Service

Efficient, reliable, and secure access to and management of information about the location of file replicas is an important service in a Data Grid.  We discuss the design and implementation of this Data Grid component in other documents.

Current status: A replica catalog API has been designed and a centralized implementation constructed; the implementation is being used in a variety of Data Grid applications and is proving sufficient for short-term requirements [2].  A requirements analysis and design for a more scalable, distributed replica location service has been developed [5].

Future plans: Develop a scalable, distributed replica location service.

## 7.4   Catalog Management Services

The issue of catalog management is complex, as the catalogs deal with both application-specific and data-grid specific entities. For example, both the derived metadata catalog (DMDC) and the MDC defined in [9] contain application-specific attributes. Applications have spent considerable

effort to develop MDCs, and it is not within the scope of this work to design the catalog schemas for applications: instead, we want to design interfaces to existing structures. Therefore it is necessary for the Data Grid architecture to:

- Utilize currently existing MDCs. Provide means of querying and updating the catalogs.

- Provide a derived data catalog (DDC) structure that captures the descriptions of transformations performed by the applications.

- Build DDC and replica catalog (RC) structures that are as generic as possible.

- Provide well-defined interfaces to the catalogs, so that both users and applications can construct relevant queries.

Current status: Initial proposal for the virtual data catalogs has been proposed [9], but needs to be further evaluated and discussed.

Future plans:

- Design mediators that can interface with the application-specific catalogs. These mediators will allow the Data Grid to retrieve information from the various catalogs already developed by the applications as well as update those catalogs upon data materialization.

- Augment the databases used by the applications with new attributes that support the Data Grid, such as the transformation used to produce the data, in the case of MDC or the ids of the derived data product, in the case of DMDC.

- In the DDC, support attributes that are common to all applications, such as transformation names and input file names. Explore the use of G-DAGs as derived data descriptors in the DDC.

### 7.5  *Grid Container Management Services(GCMS)*

GCMS will be used to associate objects to logical containers as well as to provide the ability to import/export data into the grid.

Importing into the grid is performed by writing objects and their associated attributes into a *grid container*. Accessing of objects is done via the export function of GCMS.  Obviously, GCMS is an application specific component, as it needs to understand the structure of an application object and it needs to decide on the best mapping of objects to containers for a given application.

Current Status: GCMS is still in the design stage.

- Future Plans: We need to fully understand what are the requirements on that service. Look into SRB and see how it uses containers to manage collections.

## 8   Policy and Security Services

The GSI mechanisms described in Section 4.2 address basic authentication and authorization requirements.  A set of higher-level services build on this base to address issues of policy and credential management that arise in a Grid environment.  Specifically:

- The *community authorization service* supports the specification and enforcement of the *community policies* that determine who is allowed to use what resources for what purposes.

- *Online credential repositories* and *online certificate authorities* address scalability and usability issues associated with public key infrastructure.

## 8.1  Community Authorization Service

Community authorization services [ref? – at least refer to Laura's URL for the CAS2 doc] (CAS) are an important component of the Data Grid authorization architecture.  A CAS enables resources used by a community to implement a shared global policy for use of these community resources.  Examples include enabling group access to a specific collection of files (regardless of where they are stored), or ensuring fair share use of network bandwidth across all members of the community.  The basic idea is to enable a resource owner to delegate authorization decisions to a community representative (the CAS) that specifies and enforces the global policies, such as "ingestion of new data takes precedence over analysis" or "no one ATLAS user may consume more than 30% of the compute resources dedicated to ATLAS."

A CAS architecture is under development and will be described in a separate document.  This CAS definition leverages GSI security protocols.  The central design concept is to use the CAS to issue credentials to a user that are good for a class of operations, and for other services to be able to interpret the validity of the requested operation without resorting to any other on-line service.

Current status: An early CAS prototype was demonstrated in August 2001.  Development is ongoing, and a more concrete version will be available by end of 2001.

Future plans: Complete CAS development and integrate CAS capabilities into data and compute servers.

## 8.2  PKI Scalability and Usability

An issue of ongoing investigation within the Globus project (under DOE funding) relates to the development of techniques aimed at increasing scalability and usability of public key infrastructure (PKI) systems.  For example, online certificate authorities can be used to map from local credentials (e.g., Kerberos tickets) to PKI credentials.  Online credential repositories can be used to cache credentials for subsequent use from e.g. browsers [23].  These developments are not immediately on the critical path for Data Grid systems, but will be important long term.

Current status: An online credential repository, MyProxy, exists [23].

Future plans: Track ongoing work in Globus project and GGF.

# 9   Replication

We discuss two issues here: data replication, including event data and metadata catalogs, and code distribution.

## 9.1  Replica Management Services

An effective technique for improving access speeds and reducing network loads when many users must access the same large datasets can be to replicate frequently accessed datasets at locations chosen to be "near" the eventual users.  However, organizing such replication so that it is both reliable and efficient can be a challenging problem, for a variety of reasons.  The datasets to be moved can be large, so issues of network performance and fault tolerance become important.  The individual locations at which replicas may be placed can have different performance characteristics, in which case users (or higher-level tools) may want to be able to discover these characteristics and use this information to guide replica selection.  And different locations may have different access control policies that need to be respected.

A replica management service is an entity responsible for keeping track of replicas, providing access to replicas, generating (or deleting) replicas when required. This service can clearly layer on functionality described above, specifically the Replica Catalog and Storage System access protocols and reliable high speed data movement.

We note that high-level storage management functions can be created at the collective level by composing elemental storage systems via resource level protocols. For example, hierarchal storage management can be viewed as a replication policy that manages an archival storage element and a networked storage element.

Numerous other replica management strategies can be imagined, but the following basic mechanisms appear to have general utility:

- Reliable, high-speed data movement – Ruth: I still think this is a separate service – Data Transport –above communications and below Replica Management.

- Fault tolerant and scalable Replica catalogs able to keep track of where replicas have been created and "in progress" replication efforts.

- Mechanisms for creating new replicas and removing old ones

- Mechanisms for checking on and/or enforcing the consistency of existing replicas. (How can we do this without knowledge of the internal structure of files?)

Current status: We have in place replica management functions developed within the Globus Data Grid Toolkit [2] and within the Storage Resource Broker system [http://www.npaci.edu/DICE/SRB/index.html]. Both systems support multiple data transfer protocols , These functions build on the GridFTP or other data transport protocols or application services and replica catalog mechanisms described below to provide functions for:

- The registration of files with the replica management service.

- The creation and deletion of replicas for previously registered files.

- Inquiries concerning the location and performance characteristics of replicas.

- The updating of replicas to preserve consistency when a replica is modified. (A formal definition of consistency in the Replica management services is yet to be defined).

- Management of access control at both a global and local level.

We also have in place the services provided by GDMP [18, 28], although these are very specific to a particular implementation. For example, GDMP assumes the use of Objectivity databases.. More. Other data transport  services such as bbcp (http://www.ihep.ac.cn/~chep01/paper/7-018.pdf), bbftp (http://doc.in2p3.fr/bbftp/) are used by several experiments. These  provide short term enhanced services tailored for the HENP environment.

Technical details: Pointers to be provided.

Future plans:

- Determine experimentally how effective these techniques are in practice.

- Create a *replica management service* that encapsulates these functions and supports remote requests for replica creation, etc.  (A prototype is to be shown at SC'2001 in November.)

- Develop basic ideas for replica generation/deletion services.  Work is underway at U.Chicago in this area, by PhD student Kavitha Ranganathan.

### *9.2   Code Distribution*

Secure, reliable code distribution is a critical concern for Data Grids: it must be easy to distribute a new piece of application software to a set of compute sites, and to update software as and when required.  The problem is clearly complex and multidimensional.  Publication of software installation information via MDS is important.

Current status: All physics experiments have some technologies  and a certain degree of automation to support code distribution and verification BaBar uses cvs and rshell distribution scripts; D0 uses cvs and the Fermilab ups/upd utilities; CMS uses SCRAM which is being extended to support code distribution; ATLAS uses CMT and PACMAN. EU DataGrid are looking at LCFG [3] from Edinburgh, which we should track.  Globus publishes information about installed Globus software (but not other software) via MDS.

To do: Refine requirements statement.  Survey available software.  Enlist someone to produce something.

## 10  Performance Estimation and Evaluation

## 11  Planning, Execution and Error Recovery

## 12  Missing Components

For convenience we provide here a list of the major Data Grid components identified in this document for which no implementation exists or is expected to become available in the near future.  These components represent opportunities for others to contribute to development.

Code distribution service.  See Section 9.2.

Intrusion detection service; activity logging.  See Section 7.2.

Accounting.  See Section X.

Debugging and tracing.

To do: Add to the list of missing services.

## Acknowledgments

## Bibliography

1.      RDF. http://www.w3.org/TR/REC-rdf-syntax/.
2.      Allcock, W., Bester, J., Bresnahan, J., Chervenak, A.L., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D. and Tuecke, S., Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing. In *Mass Storage Conference*, (2001)

3. Anderson, P. and Scobie, A., Large Scale Linux Configuration with LCFG. In *4th Annual Linux Showcase and Conference*, (2000). [www.usenix.org/publications/library/proceedings/als2000/full_papers/anderson/anderson.pdf](www.usenix.org/publications/library/proceedings/als2000/full_papers/anderson/anderson.pdf)

4. Baru, C., Moore, R., Rajasekar, A. and Wan, M., The SDSC Storage Resource Broker. In *Proc. CASCON'98 Conference*, (1998)

5. Chervenak, A., Foster, I., Iamnitchi, A. and others. A Scalable Replica Location Service. Work in progress., 2001.

6. Chervenak, A., Foster, I., Kesselman, C., Salisbury, C. and Tuecke, S. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Data Sets. *J. Network and Computer Applications* (23). 187-200. 2001.

7. Collaboration, T.G. The GriPhyN Project. [www.griphyn.orgTBD.](www.griphyn.orgTBD.), 2000.

8. Czajkowski, K., Fitzgerald, S., Foster, I. and Kesselman, C., Grid Information Services for Distributed Resource Sharing. In *10th IEEE International Symposium on High Performance Distributed Computing*, (2001), IEEE Press, 181-184

9. Deelman, E., Foster, I., Kesselman, C. and Livny, M. Representing Virtual Data: A Catalog Architecture for Location and Materialization Transparency. 2001.

10. Forrest, S., Hofmeyr, S.A. and Somayaji, A. Computer Immunology. *Communications of the ACM*, *40* (10). 88-96. 1997.

11. Foster, I. and Kesselman, C. A Data Grid Reference Architecture. GriPhyN 2001-6, 2001, [www.griphyn.org/document-server](www.griphyn.org/document-server).

12. Foster, I. and Kesselman, C. (eds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

13. Foster, I., Kesselman, C., Tsudik, G. and Tuecke, S. A Security Architecture for Computational Grids. In *ACM Conference on Computers and Security*, 1998, 83-91.

14. Foster, I., Kesselman, C. and Tuecke, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, *15* (3). 200-222. 2001. [www.globus.org/research/papers/anatomy.pdf](www.globus.org/research/papers/anatomy.pdf).

15. Foster, I., Roy, A. and Sander, V., A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *Proc. 8th International Workshop on Quality of Service*, (2000)

16. Frey, J., Tannenbaum, T., Foster, I., Livny, M. and Tuecke, S., Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *10th International Symposium on High Performance Distributed Computing*, (2001), IEEE Press, 55-66

17. Gullapalli, S., Czajkowski, K., Kesselman, C. and Fitzgerald, S. The Grid Notification Framework. Global Grid Forum, Draft GWD-GIS-019, 2001.

18. Hoschek, W., Jaen-Martinez, J., Samar, A., Stockinger, H. and Stockinger, K., Data Management in an International Data Grid Project. In *International Workshop on Grid Computing*, (2000), Springer Verlag Press

19. Litzkow, M. and Livny, M. Experience With The Condor Distributed Batch System. In *IEEE Workshop on Experimental Distributed Systems*, 1990.

20. Livny, M. DAGMAN reference.

21. Moore, R., Baru, C., Marciano, R., Rajasekar, A. and Wan, M. Data-Intensive Computing. In Foster, I. and Kesselman, C. eds. *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999, 105-129.

22. Mukherjee, B., Heberlein, L.T. and Levitt, K.N. Network Intrusion Detection. *IEEE Network*, *8* (3). 26-41. 1994.

23. Novotny, J., Tuecke, S. and Welch, V., An Online Credential Repository for the Grid: MyProxy. In *10th IEEE International Symposium on High Performance Distributed Computing*, (2001), IEEE Press, 104-111

24.     Sander, V., Adamson, W., Foster, I. and Roy, A., End-to-End Provision of Policy Information for Network QoS. In *10th IEEE International  Symposium  on High Performance Distributed Computing*, (2001), IEEE Press, 115-126

25.     Sander, V., Foster, I., Roy, A. and Winkler, L., A Differentiated Services Implementation for High-Performance TCP Flows. In *TERENA Networking Conference*, (2000)

26.     Shoshani, A., Bernardo, L.M., Nordberg, H., Rotem, D. and Sim, A. Storage Management for High Energy Physics Applications. In *Computing in High Energy Physics 1998 (CHEP 98)*, 1998.

27.     Sompel, H.V.d. and Lagoze, C. The Open Archives Initiative Protocol for Metadata Harvesting. The Open Archives Initiative, 2001, http://www.openarchives.org/OAI_protocol/openarchivesprotocol.html.

28.     Stockinger, H., Samar, A., Allcock, W., Foster, I., Holtman, K. and Tierney, B., File and Object Replication in Data Grids. In *10th IEEE Intl. Symp. on High Performance Distributed Computing*, (2001), IEEE Press, 76-86

29.     Szalay, A. and Gray, J. The World-Wide Telescope. *Science*, *293*. 2037-2040. 2001.

30.     Tierney, B., Johnston, W., Crowley, B., Hoo, G., Brooks, C. and Gunter, D., The NetLogger Methodology for High Performance Distributed Systems   Performance Analysis. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*, (1998)

31.     Wolski, R. Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, Portland, Oregon, 1997.